型を考慮したcode2vecによるソースコードの分散表現獲得方法の考察

M2021SE009 大鷹弘史
指導教員：蜂巣吉成

1 はじめに

word2vec や doc2vec など，自然言語を入力とした分散表現の獲得方法が提案され，様々に応用されている．Alon らはソースコードのコード片の分散表現を生成する手法 code2vec[1] を提案している．code2vec は Java のメソッドのコード片からメソッド名を推測することができる．[1] では比較的高い精度でメソッド名が推測できることが述べられているが，学習データの識別子名に依存しており，不適切な識別子名を用いると精度が低下する．

プログラミング学習者は適切な変数名をつけられないことがある．オンラインジャッジシステム (OJS) などで学習者のソースコードを入手しやすくなったが，これらに code2vec を適用しても十分な精度が得られないことがある．一般に OJS ではテストによりプログラムが正解か判定するが，正解とされたコードは変数名が不適切でも変数の型や計算ロジックなどは適切であると考える．

本研究では学習者のソースコードから適切に分散表現を得るために，変数名を型に置き換えて code2vec を適用する方法について考察する．OJS で正解とされたコードに対して，変数名を置き換えたものと置き換えないもので，精度を比較する．

2 関連研究

2.1 code2vec

Alon らは，コード片の分散表現を作成する手法 code2vec[1] を提案した．word2vec が単語から分散表現を生成するように，code2vec は Java のメソッドの本体とメソッド名から分散表現を生成する．メソッドから抽象構文木（AST）を作成し，AST の任意の 2 つの終端記号とそれらを繋ぐ非終端記号を連結したものをパスと呼び，それらを特徴量とする．メソッド名の推測は特徴量の値が近いものが確率として返される．

code2vec の問題点として変数名への依存が挙げられる．code2vec は適切な命名をしていると想定される GitHub 上の Java を用いた OSS のソースコードを基として学習している．これらの識別子名に依存しており，情報量の少ない識別子名，難読化，敵対的な識別子名が与えられた場合，予測精度が低くなることがある．

2.2 CodeNet

CodeNet[2] は IBM が提供している AI を用いたソフトウェア開発支援のためのオープンソースのデータセットである．このデータセットには 50 以上のプログラミング言語で作成された約 14,000,000 のコードで構成されており，AI 向けのソースコードのデータセットとしては最大規模のものである．内容は AtCoder と AIZU Online Judge の 2 つの競技プログラミングサイトにおいて出題された問題に対する解答が含まれている．データセットは問題に対して提出されたソースコードと，問題ごとに提出されたソースコードの一覧の CSV ファイル，及び問題の一覧の CSV ファイルがある．問題に対して提出されたソースコードの中には，正誤確認用のテストケースをすべてパスしたコード以外にも，一部のテストケースをパスできなかったコード，コンパイルエラーや実行時間制限超過，メモリ制限の超過などを起こしたコードも含まれている．

3 型を考慮した code2vec の考察

本研究では，2.1 節で挙げた問題点の変数名への依存に着目し，OJS の Java ソースコードを対象として変数の型を考慮して code2vec を適用し，メソッド名とその本体のコードを学習させ，適切な分散表現を得る手法を考察する．提案方法の基本的なアイディアを述べる．

Main クラス・main メソッド

main の部分を「p+問題の番号を a を 0，z を 25 とした 26 進数表記 (3 桁)」にする

`public class Main` → `public class pfex`

`public static void main`

→ `public static void pfex`

フィールド (メンバ変数)

型に f+番号をつけて識別子名とする

`int a, b;` → `int int_f1, int_f2;`

`double x, y;` → `double double_f1, double_f2;`

局所変数

型に番号をつけて識別子名とする

`int a, b;` → `int int1, int2;`

`double x, y;` → `double double1, double2;`

Main クラス，main メソッドのクラス名及びメソッド名を変更する理由は，簡単な問題では学習者のコードが main メソッドにすべての処理を書くケースが多く，予備実験として未変更の状態のモデルでメソッド名の推測を行ったところ，結果がすべて main と返されたので，このような処理をすることとした．処理を行うことで問題ごとの特徴をモデルに学習させ，妥当なメソッド名を返す確率が上がると考えた．

変数を変更する理由は学習者は変数を a や n などの特に意味のないものにすることが多いと考えたからである．モデルに変数の型の情報を埋め込むことで，適切なメソッド名を返す確率が上がると考えた．

4 実験

実験目的は次のとおりである．

目的 1 提案方法を適用したデータセットと適用しないデータセットでの結果の違いを確認する

目的 2 データセットは元のコードのメソッド名を変更したものを入力して，メソッド名を推測できるか

目的 3 OJS の正解と不正解のコードを判別できるか

4.1 実験方法

CodeNet の中から AtCoder の問題 317 問に対して正解の Java コード (Accepted)23841 個のうち，1000 個をテスト，2000 個を検証，それ以外を学習用に用いた．コード群に対して，コード内の Main クラス，main メソッドのクラス名及びメソッド名を「p09876」のように CodeNet 内で振られていた問題番号に変更し，さらに情報の欠落を防ぐため問題番号を a を 0，z を 25 とした 26 進数表記に変更する (例:p02555→ pduh)．これを処理 1 とする．コード内の int 型，double 型の変数名をそれぞれフィールドは「`int_fn`」「`double_fn`」，局所変数は「`int_n`」「`double_n`」(**n** は連番）に変更する．これを処理 2 とする．

実験に使う学習したモデルの妥当性を確認するために，正解のコード群は未処理，処理 1 まで，処理 2 まで，行った状態の 3 状態をそれぞれ code2vec に適用し，code2vec 内で検証を行い，それぞれで最も高かった Accuracy，Precision，Recall，F1 の値について比較を行う．

4.1.1 実験 1

適切でない識別子名を型に変えたことの有効性を確認するために，識別子名が適切でないメソッドについて，メソッド本体のコードからメソッド名が推測できるか，提案方法を適用したデータセットと適用しないデータセットでの結果の違いを確認する．

正解の各コード群においてテスト用のデータセット内から抽出した 10 個の main メソッドを入力として用いて推測を行う．なお，メソッド名は「main」を「aaa」や「ccc」の様に変更する．ここで，行数の少ないコードは main のみが記載されており，識別子名が適切でないと判断したため，抽出の基準は 20 行以下のコードで，一度選ばれた問題は選ばれないようにランダムに抽出した．未処理のモデル (以下モデル 0) と処理 1 を適用したモデル (以下モデル 1) には入力として識別子名を変更していないメソッドを，処理 1 と処理 2 を適用したモデル (以下モデル 2) には入力として識別子名を変更したメソッドを用いる．予想としては，提案方法の方が適切にメソッド名を推測できると考える．実験で用いたメソッドの問題番号は 2555，2611，2819，2841，3030，3048，3328，3470，3543，4044 の 10 問である．

例として問題番号 2555 を List1 に示す．

List 1 データセット内の Java メソッド (問題番号 2555)

```java
public static void aaa(String[] args) {
  Scanner sc = new Scanner(System.in);
  int s= sc.nextInt();
  long div=1_000_000_007;
  long[]ans=new long[s+1];

  for (int i = 1; i <= s; i++) {
    if(i<=2)      ans[i]=0;
    else if(i==3) ans[i]=1;
    else          ans[i]=(ans[i-1]+ans[i-3])%
      div;
  }
  System.out.println(ans[s]);
}
```

4.1.2 実験 2

適切な識別子名を型に変えたことの影響を確認するために，識別子名が適切なメソッドについて，メソッド本体のコードからメソッド名が推測できるか提案方法を適用したデータセットと適用しないデータセットでの結果の違いを確認する．データセット内の main ではない適切な名前を持つメソッドを 5 つ取り出して入力として用いる．なお，メソッド名は「AAA」「BBB」のように変更する．前述の実験と同様に，モデル 0 とモデル 1 には入力として識別子名を変更していないメソッドを，モデル 2 には入力として識別子名を変更したメソッドを用いる．予想としては，提案方法の方が適切なメソッド名の推測確率が落ちると考える．

元のメソッド名はそれぞれ「binarySearch」「compareLower」「sum」「inverse」「radixSort0」である．例として「binarySearch」を List2 に示す．

List 2 メソッド (binarySearch)

```java
static private int binarySearch(long num, long
  [] orderedArray){
    int lowerBorder = -1;
    int upperBorder = orderedArray.length;
    int mid;

    while(upperBorder - lowerBorder >1) {
        mid = (upperBorder + lowerBorder)/2;
        if(orderedArray[mid]<=num) {
            lowerBorder = mid;
        }else {
            upperBorder = mid;
        }
    }
        return lowerBorder;
}
```

4.1.3 実験 3

正解のコードと不正解のコードの特徴をとらえて，それらが区別できるか確認するために，正解のコードに不正解のコードの 17773 個を含め 2000 個をテスト，4000 個を検証，それ以外を学習用に用いたたデータセットを作成した．不正解のコードは正解と区別するため main メソッドを「p—」ではなく「e—」のような形に変更した入力として用いたコードは識別子名を変更したメソッドである．実験で用いたメソッドの問題番号は正解のコードが 2555，2611，2819，2841，3030 の 5 問，不正解のコードが 2841，3267，3817，3861，4044 の 5 問である例として不正解コード問題番号 4044 を List3 に示す．

List 3 不正解メソッド (問題番号 4044)

```java
  public static void efzo(String[] args) {
  Scanner scan = new Scanner(System.in);
  int int_1 = scan.nextInt();
  int int_2 = scan.nextInt();
  Set<String> set = new TreeSet<String>();
  for(int i = 0; i < int_1; i++) {
    set.add(scan.next());
  }
```

```
    for(String s:set) {
      System.out.print(s);
    }
    System.out.println();
  }
```

4.2 実験結果

それぞれのモデルに対して code2vec 内の検証 (Evaluating) を行い，出力された各モデルそれぞれの Accuracy，Precision，Recall，F1 の値について表 1 に示す．どのモデルにおいても十分に学習が行われた．

表 1 Accuracy，Precision，Recall，F1 の値

	Accuracy	Precision	Recall	F1
モデル 0	0.974	0.943	0.942	0.943
モデル 1	0.930	0.900	0.897	0.898
モデル 2	0.952	0.923	0.921	0.922

4.2.1 実験 1

データセット内のコードを 10 個入力として用いた推測結果を示す．それぞれのパターンにおいて，モデル 0 では 0.999999 以上の確率で「main」と推測していた．モデル 1 とモデル 2 について入力として与えられた問題番号がどの順位で推測されたかまとめた表を表 2 に示す．

表 2 モデル 1 とモデル 2 の推論結果（丸括弧内は確率の平均）

	モデル 1	モデル 2
1 位	5 回 (0.9862)	6 回 (0.9194)
2 位	1 回 (0.2654)	1 回 (0.2726)
3 位	1 回 (0.0164)	0 回
4 位	0 回	1 回 (0.1026)
5 位	0 回	1 回 (0.0246)
7 位	1 回 (0.0065)	0 回
推測失敗 (10 位〜)	2 回	1 回

推測結果は入力としてメソッドを与えた際に，「入力メソッドの名前は右の角括弧内の名称である確率が丸括弧内の数字である」というものを上から 10 位の予想まで示したもので返される．例として List1 の問題番号 2555(pduh) の main メソッドをモデル 1 に適用した結果を図 1 に，モデル 2 に適用した際の結果を図 2 に示す

```
Original name:  aaa
    (0.510005) predicted: ['pfgi']
    (0.265440) predicted: ['pduh']
    (0.108204) predicted: ['peel']
    (0.047100) predicted: ['pefh']
    (0.020727) predicted: ['pfjo']
    (0.016948) predicted: ['pfmv']
    (0.013371) predicted: ['pfgj']
    (0.009945) predicted: ['perq']
    (0.004665) predicted: ['pfjm']
    (0.003595) predicted: ['pfjn']
```

図 1 推測結果（問題番号 2555(pduh)，モデル 1）

```
Original name:  aaa
    (0.674551) predicted: ['pduh']
    (0.262890) predicted: ['pfgi']
    (0.017293) predicted: ['pefl']
    (0.015130) predicted: ['peel']
    (0.010533) predicted: ['pecn']
    (0.004577) predicted: ['pfos']
    (0.004410) predicted: ['pdzh']
    (0.004212) predicted: ['pepo']
    (0.003244) predicted: ['pepc']
    (0.003161) predicted: ['peeh']
```

図 2 推測結果（問題番号 2555(pduh)，モデル 2）

4.2.2 実験 2

コード内の main ではないメソッドを 5 つ取り出して入力として用いた結果を示す．各メソッドを入力としたモデルごとの 1 位の推測をまとめた表を表 3 に示す．

表 3 実験 2 の結果（丸括弧内は確率）

	モデル 0	モデル 1	モデル 2
binarySearch	binarySearch (0.9837)	binarySearch (0.9988)	binarySearch (0.9988)
compareLower	compareLower (0.9999)	compareLower (0.9997)	compareLower (0.9312)
sum	sum (0.9984)	sum (0.9788)	sum (0.9406)
inverse	inverse (0.7245)	inverse (0.9999)	inverse (0.9999)
radixSort0	fill (0.5840)	addEdge (0.2917)	min (0.2099)

例として List2 の「binarySearch」を入力として用い，モデル 0 で推測した結果を図 3，モデル 1 で推測した結果を図 4，モデル 2 で推測した結果を図 5 に示す．

```
Original name:  aaa
    (0.983706) predicted: ['binary', 'search']
    (0.012360) predicted: ['upper', 'bound']
    (0.001898) predicted: ['lower', 'bound']
    (0.000733) predicted: ['upperbond']
    (0.000615) predicted: ['bs']
    (0.000215) predicted: ['bin', 'search']
    (0.000180) predicted: ['bfs']
    (0.000145) predicted: ['bi', 'search']
    (0.000079) predicted: ['update']
    (0.000069) predicted: ['beam', 'search']
```

図 3 推測結果（メソッド名「binarySearch」，モデル 0）

```
Original name:  aaa
    (0.998834) predicted: ['binary', 'search']
    (0.000306) predicted: ['upper', 'bound']
    (0.000237) predicted: ['bi', 'search']
    (0.000110) predicted: ['bisect']
    (0.000105) predicted: ['bi', 'search', 'max']
    (0.000097) predicted: ['get', 'min']
    (0.000087) predicted: ['get', 'max']
    (0.000082) predicted: ['pfpi']
    (0.000077) predicted: ['next', 'double']
    (0.000065) predicted: ['get', 'max', 'cost']
```

図 4 推測結果（メソッド名「binarySearch」，モデル 1）

```
Original name:  aaa
    (0.998771) predicted: ['binary', 'search']
    (0.000244) predicted: ['pevd']
    (0.000187) predicted: ['up']
    (0.000170) predicted: ['pdud']
    (0.000144) predicted: ['bisect']
    (0.000106) predicted: ['pfli']
    (0.000105) predicted: ['search']
    (0.000096) predicted: ['shortest', 'path']
    (0.000091) predicted: ['execute']
    (0.000087) predicted: ['peng']
```

図 5 推測結果（メソッド名「binarySearch」，モデル 2）

4.2.3 実験 3

実験 3 の 1 位の推測結果，問題番号の推測をした順位とその確率を表 4 に示す.

表 4 実験結果 (実験 3)

問題番号	1 位の推測	問題番号を推測した順位
pduh	pfgi(0.3227)	4 位 (0.1118)
pdwl	eevd(0.5330)	推測失敗 (10 位〜)
peel	peel(0.9593)	1 位 (0.9593)
pefh	pefh(0.9883)	1 位 (0.9883)
pemo	pfsf(0.8332)	5 位 (0.0109)
eefh	pefh(0.9837)	4 位 (0.0010)
eevr	pfsf(0.7930)	推測失敗 (10 位〜)
efqv	pfqv(0.9981)	2 位 (0.0015)
efsn	efsn(0.6481)	1 位 (0.6481)
efzo	pfzo(0.7683)	3 位 (0.0640)

5 考察

表 1 の Accuracy，Precision，Recall，F1 の値について，どのモデルも高い値を示しており，十分に学習が行われたと考える．特に，モデル 0 がいずれも良い値を示した．これは，モデル 0 に関しては，main が最も確率が高くなるように推測を返すようになっているためであると考える．

正解の各コード群を入力として用いた結果の考察としては，モデル 0 ではいずれの問題においてもほぼ 1.0 に近い精度で main が推測結果となっているが，main メソッドを入力として用いたのでこの結果は妥当ではあるが，どの問題の main メソッドであるかが不明瞭であるので，有用なモデルとは言えない．モデル 1 とモデル 2 で比較すると以下 3 パターンに分類できる．

1. モデル 1 のほうが高い確率で問題番号を推測できているケース
   問題番号　2819(eel)，2841(efh)，3328(eya)，3470(fdm)，4044(fzo)
2. モデル 2 のほうが高い確率で問題番号を推測できているケース
   問題番号　2555(duh)，3030(emo)，3048(eng)，3543(fgh)
3. どちらも問題番号を推測できていないケース
   問題番号 2611(dwl)

1. のパターンはいずれもモデル 1 が 0.94 以上の確率で候補に挙げているが，モデル 2 も高い確率で推測をしている．一方，2. のパターンではモデル 2 は最も高い確率で推測をしているわけではないが，モデル 1 の推測確率と比較すると，かなり高い確率で推測ができており，モデル 2 の方が有用であると考える．

コード内の main でないメソッドは 3 つのモデルで 1 個のコード (radixSort0) を除いて高確率で正しい推測ができた．radixSort0 が推測できなかった理由はこの名前が使われているファイルが 1 個しかないためであると考える．

実験 3 について，結果を分析するために 1 位の推測結果をまとめた表を表 5 に示す.

表 5 結果まとめ (実験 3)

	正解か不正解かは正しい (1)	正解か不正解かを間違えた (2)
問題は正しい (A)	peel, pefh, efsn	eefh, efqv, efzo
問題を間違えた (B)	pduh, pemo	pdwl,eevr

上記の表 5 から，2A（問題番号は正しいが，正解か不正解かを間違えた）のパターンには不正解のコードのみが入っている．また，表 4 の結果から，2A のパターンは正解の問題番号は高い確率だが，不正解の問題番号は低い確率になっている．2B（問題番号も，正解か不正解かも間違えた）のパターンは問題番号を推測することに失敗している．2 のパターンでは，問題を正しく推測できたか否かに関わらず，正しい問題番号の推測確率が 0.07 以下と低くなっている．実験 1 のモデル 2 と実験 3 のモデルの正解コードを入力として用いたものを比較すると，実験 3 の 2B のパターンは実験 1 でも推測結果が悪いことがわかった．このことから，正解か不正解を正しく判断できなかった問題は総じて推測結果が悪い．これを改善することが今後の課題であると認識した.

6 おわりに

本研究では，学習者のソースコードから適切な分散表現を得るために，変数名を型に置き換えて code2vec を適用する方法の考察を行った．OJS で正解とされたコードに対して，変数名を置き換えたものと置き換えないもので，精度を比較した．正解の各コード群を入力として用いた結果として，モデル 1 の方が高い確率で推測したパターンでは，モデル 2 においても高い確率で推測しているが，モデル 2 の方が高い確率で推測したパターンでは，モデル 1 は推測できても比較して低い確率か，あるいは推測できていない．以上のことから提案方法で識別子名を変更し，code2vec への適用を行うことでより良い推測を出来るようになったと言える．今後の課題として，正解のコードと不正解のコードの特徴を表現，区別できるような前処理方法の考察が挙げられる.

参考文献

[1] Uri Alon, et al.:“code2vec: Learning Distributed Representations of Code”, Proc. ACM Program. Lang. Vol. 3, January 2019. https://code2vec.org/

[2] Ruchir Puri, et al.:“CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks” Conference on Neural Information Processing Systems 2021, May 2021.